Appendix B

DATA MAPPING AND ROTATION FUNCTIONS FOR THE MASSIVELY PARALLEL PROCESSOR

Anthony P. Reeves and Cristina H. Francfort de Sellos Moura

School of Electrical Engineering
Cornell University
Ithaca, New York 14853

The Massively Parallel Processor is a SIMD computer with 16384 processing elements connected in a 128 x 128 mesh. Such an organization is ideal for problems which involve near neighbor iterations, but for other problems which involve other data mappings it is often considered to be inefficient. In this paper a general algorithm for implementing arbitrary permutations and mappings on such systems is presented. Efficient matrix rotation algorithms based on this permutation function are also discussed. Nearest neighbor, bilinear interpolation and bicubic spline interpolation schemes are considered. These algorithms are extended for the case when the matrix to be processed is larger than the parallel hardware dimensions.

INTRODUCTION

A convenient way to interconnect a very large number of processors is in a two dimensional grid or mesh; this interconnection arrangement is very simple to implement, has a cost which increases linearly with the number of processors and is very suitable for a large number of algorithms. An example of such a system is the Massively Parallel Processor [1] which involves 16384 bit-serial processors organized in a 128 x 128. The MPP is programmed in a high level language called Parallel Pascal [2].

The only permutation function which is directly implemented by the MPP is the near neighbor rotate (or shift). The direction of the rotation may be in any of the four cardinal directions. In Parallel Pascal the main permutation functions are multi-element rotate and shift functions; other permutations are built on these primitives.

The rotate function takes as arguments the array to be shifted and a displacement for each of the arrays dimensions. For example consider a one dimensional array a specified by

```
a: array [0..n] of integer;
```

The rotate statement

```
a := rotate( a, 5);
```

is equivalent to

```
for i := 0 to n do
    a[i] := a[(i + 5) mod (n + 1)];
```

The rotation utilizes the toroidal end around edge connections of the mesh. The *shift* function is similar except that the mesh is not toroidally connected and zeroes are shifted into elements at the edge of the array; therefore, the shift function is not a permutation function in the strict sense. The concept of the rotate and shift functions extend to n dimensions; on the MPP the last two dimensions of the array correspond to the parallel hardware dimensions and are executed in parallel, higher dimension operations are implemented in serial. The cost of the rotate function is dependent on the distance rotated. It also depends on the size of the data elements to be permuted.

There is no simple known way to decompose an arbitrary permutation into a minimal sequence of operations on an MPP like system. In this paper a heuristic algorithm is described. The algorithm exploits the local consistency of data which occurs in many practical applications. An effective application of this algorithm to matrix rotation is presented. For some permutations, such as the perfect shuffle, which do not directly exhibit this consistency property, the algorithm may not be very effective.

For many applications the physical dimensions of the parallel hardware are smaller than the dimensions of the array to be processed. In this case the data array is processed as a set of blocks. An extension of the permutation algorithm to deal with this situation is discussed.

The program and algorithm examples given in this paper use the Parallel Pascal notation. This notation involves three extensions to standard Pascal:

1) expressions involving whole arrays are permitted;
2) the **where - do - otherwise** control statement is available. This statement is a parallel version of the **if - then - else** statement; the control expression must evaluate to a Boolean array. All array assignments within the controlled statements must be conformable with the control array and are masked by it.
3) the functions *any* and *min* are the array reduction functions or and minimum respectively.

MATRIX PERMUTATIONS

The matrix permutation algorithm presented in this paper is a general algorithm for implementing arbitrary permutations of a two dimensional matrix on mesh connected parallel processors. It is also capable of performing any onto mapping. It uses a heuristic approach to reduce the execution time.

The permutation of a matrix a is specified by two coordinate matrices c and r which have similar dimensions as a. For a matrix element b[i,j] the corresponding elements r[i,j] and c[i,j] specify the row and column indices respectively of where the related element of a is located. That is, the permutation is specified by

$$b[i,j] := a[\, r[i,j], c[i,j]\,]$$

More formally, the data arrays involved in the permutation are specified by:

```
a,b : array [1..nrow,1..ncol] of data;
{where data is any base type}
r : array [1..nrow,1..ncol] of 1..nrow;
c : array [1..nrow,1..ncol] of 1..ncol;
```

In order to compute the relative distance that the data must be moved, two pixel element identifying matrices *idr* and *idc* are precomputed. They contain the following:

```
idr[i,j] := i;
idc[i,j] := j;
for all i,j
```

The relative distances to be moved are then specified by

```
rr := (r-idr) mod nrow;
rc := (c-idc) mod ncol;
```

In a permutation the data may be shifted in any of the four quadrants in order to reach a specified destination. However, in the following algorithms only positive data shifts are considered, i.e. in the up and left directions. The other three quadrants are covered by using modulo arithmetic for shift distance calculations and implementing data movement with the rotation function which utilizes the end around mesh connections. We have investigated a modified heuristic algorithm which checks all four quadrants and moves in the optimal direction. Fewer data shift operations are required but the overhead due to checking alternative directions is significantly higher.

A simple permutation algorithm

A simple naive algorithm to achieve an arbitrary permutation is to slide a over all the possible positions of b, assigning the specified elements of a to each element of b when they are in the correct position.

```
for i:= 1 to nrow do
    begin
      for j:= 1 to ncol do
          begin
            where (rr = i) and (rc = j) do
                    b := a;
            a := rotate(a, 0, 1);
          end;
      a := rotate(a, 1, 0);
    end;
```

This algorithm involves $O(n^2)$ operations for an n x n matrix.

The Heuristic algorithm

In many permutations which occur in practice there are well defined patterns for the data. For example, near neighbor shifts are trivial with complexity $O(1)$, perfect shuffles can be implemented in $O(n)$ time. The heuristic algorithm attempts to take advantage of the fact that rr and rc will be the same or similar for many elements. This is particularly true for operations such as matrix warping.

The algorithm first slides (rotates) a as many locations up and left as possible such that future backtracking will not be necessary. If any element of a is correctly positioned over b (i.e. $(rr = 0)$ and $(rc = 0)$) then b is updated. Otherwise, atr, which is a copy of the current version of a, is slid in the upwards direction until all outstanding elements of b, for which the current $rc = 0$, are satisfied. The algorithm then shifts as far as possible up and left again and repeats the above procedure until all elements of the result mask are false, i.e. b is complete.

The following variables are used in the algorithm :

Variable declaration
- *mask,masktr* : array [1..nrow, 1..ncol] of boolean;
- *atr* : array [1..nrow, 1..ncol] of data;
- *rrt* : array [1..nrow, 1..ncol] of 0..nrow;
- *ri, rit, lastrit* : 0..nrow;
- *ci* : 0..ncol;

Variable functions:
- *mask* : the result mask, true values indicate elements of b which have not yet received the correct element of a.
- *ri, ci* : row and column distances for the up-left move.
- *masktr* : a version of *mask* to process one column.
- *rit* : a version of *ri* used to process one column.
- *atr* : a version of a used to process one column.
- *rrt* : a version of *rr* used to process one column.
- *lastrit* : the last value of *rit*.

The Parallel Pascal version of the heuristic algorithm is as follows:

```
lastrit :=0;
b := a;
mask := (rr <>0) or (rc <>0);

while any(mask, 1 ,2) do
   begin  { iterate until the permutation is complete }
      ri := min(rr, 1, 2);
      ci := min(rc, 1, 2);
      a := rotate(a, ri, ci);  { move up and left as far as possible }
      rr := rr - ri;
      rc := rc - ci;
      masktr := (rr = 0) and (rc = 0);
      if any(masktr, 1, 2) then  { satisfy elements for the
                                   current position }
         atr := a
      else
         begin     {satisfy each element for the given column}
            where rc = 0 do
               rrt := rr
            otherwise
               rrt := nrow;
            rit := min(rrt, 1, 2);
            masktr := rrt = rit;
            { the next seven statements implement }
            { the statement atr = rotate (a, rit, 0) }
            { but also take advantage of the previous shifts  }
            if ci <>0 then
               begin
                  atr := a;
                  lastrit := 0;
               end;
            atr := rotate( atr, rit - lastrit, 0 );
            lastrit := rit;
         end;
       where masktr do {update b for the current location of a}
          begin
             b := atr;
             rr := nrow;
             rc := ncol;
             mask := false;
          end;
   end;
```

This algorithm is bounded by n^2 iterations. However, this must be considered a loose bound since we currently do not know a permutation which would require all n^2 iterations. The algorithm requires one iteration for a positive single element shift permutation but n-1 operations for a negative shift since the rotate is in the wrong direction.

Algorithm Cost

The cost of the naive algorithm is proportional to the number of rotate operations, i.e. n^2 * (the cost of a one element rotate operation plus two comparison operations). The heuristic algorithm has two major cost components : the rotate operations as noted before and the (min) reduction functions. The reduction functions are used to compute the multi-element distance for moves. In the tables for the performance of the algorithm, both the total number of element rotates and the total number of

reduction operations are given.

The relative cost of a rotation and reduction is both system and data size dependent. For the MPP, the cost of a reduction function is in the order of 4.2 μs whereas the single element rotation of 32-bit data requires in the order of 3.2 μs to 9.6 μs depending upon the number of successive rotate operations. Therefore, the reduction functions may represent a significant portion of the computation cost. With careful low level programming the reduction operations can be overlapped with data rotate operations such that their effective cost is in the order of 1.4 μs. If the MPP was augmented with a small amount of additional hardware similar to that outlined in [3] then the reduction time could be reduced to 1.5 μs over half of which could could be overlapped with data rotation operations. The heuristic algorithm always requires less iterations and rotations than the naive algorithm; however, the additional overhead of the reduction function may make it less efficient in some instances.

Permutation Results

The results of some permutations performed in order to obtain rotated matrices of size 32 x 32 are given in Table I and II. These rotations are into mappings rather than permutations (see the matrix rotation section for details). For comparison, the naive algorithm requires 1024 iterations, 1024 rotate operations and zero reductions for any 32 x 32 matrix permutation or mapping. Table III contains the results for perfect shuffle permutations for different size matrices. The result for perfect and inverse shuffles are identical for any matrix size.

TABLE I: Cost for a near neighbor rotation on a 32 x 32 matrix centered at 16 16.

Angle of rotation	Matrix rotation mapping cost		
	iterations	rotations	reductions
0	0	0	0
15	124	562	340
30	262	620	748
45	505	837	1464
60	741	1022	2163
75	724	1019	2119
90	528	1007	1552

TABLE II: Cost for a near neighbor rotation on a 32 x 32 matrix centered at 1 1.

Angle of rotation	Matrix rotation mapping cost		
	iterations	rotations	reductions
0	0	0	0
15	235	304	666
30	437	517	1266
45	625	683	1825
60	779	829	2292
75	889	956	2631
90	993	992	2946

The perfect shuffle is an example of a permutation which does not exhibit the locality property. The number of algorithm iterations needed to implement shuffles directly is $(n-1)^2$. However, the separability property of the two dimensional shuffle is not being used. If we use the permutation algorithm to the permutation in two stages, i.e. first shuffle the rows and then shuffle the columns, then $n-1$ iterations are needed for each permutation. Therefore, the perfect shuffle when implemented directly has complexity $O(n^2)$, but when computed in two stages the algorithm is much more effective and has $O(n)$ complexity. The results of implementing the perfect shuffle as two separable shuffles are also given in Table III. Table IV shows the results of a random permutation; this demonstrates that the heuristic is not effective when the mapping does not posses the locality property.

TABLE IV: Permutation cost for a random permutation.

matrix size	Random Permutation		
	iterations	rotations	reductions
32 x 32	630	995	1851

LARGE ARRAYS

Frequently the data to be processed by a parallel processor will be in the format of arrays which exceed the fixed range of parallelism of the hardware. Therefore, it is necessary to have special algorithms that will deal with large arrays by breaking them down into blocks manageable by the hardware, without loosing track of the relationships between different blocks.

One scheme, which is frequently used on the MPP, is to partition the large array into blocks which are conveniently stored in a four dimensional array. The range of the first dimension of this array specifies the number of blocks in each row of the large matrix and the range of the second dimension specifies the number of blocks in each column. Given a conceptual large matrix

$$mx : \textbf{array}\ [0\_x, 0\_y]\ \textbf{of}\ btype;$$

which is to be stored in an array a of type

$$\textbf{array}\ [1\_n, 1\_m, 1\_p, 1\_q]\ \textbf{of}\ btype;$$

Element i,j of the large matrix is mapped into the array a as specified by

$$mx[i,j] = a\ [1+i\ \textbf{div}\ p, 1+j\ \textbf{div}\ q, 1+i\ \textbf{mod}\ p, 1+j\ \textbf{mod}\ q]$$

For example, a 512 x 256 matrix could be stored in eight blocks as:

$$la : \textbf{array}\ [1\_4, 1\_2, 1\_128, 1\_128]\ \textbf{of}\ \textbf{real};$$

This data structure allows blocks to be manipulated independently. However, it still preserves the positional relationships of those blocks in the original large matrix.

To simplify the manipulation of large arrays on the MPP, two Parallel Pascal library functions *lrotate* and *lshift* have been developed. These functions take an array argument and two dis-

TABLE III: Cost for perfect shuffle permutations for different matrix sizes.

matrix size	Direct Shuffle Cost			Separable Shuffle Cost		
	iterations	rotations	reductions	iterations	rotations	reductions
4 x 4	9	12	22	6	6	6
8 x 8	49	56	134	14	14	14
16 x 16	225	240	646	30	30	30
32 x 32	961	992	2822	62	62	62

placement arguments, like the primitive matrix rotate and shift functions, however, in this case the array argument is a four dimensional array which is treated like a conceptually large matrix.

Many programs can be converted to operate on blocked rather than conventional matrices by simply replacing all instances of rotate and shift with lrotate and lshift respectively. This is true for the permutation programs presented; however, in the case of the heuristic permutation algorithm, this is not a very efficient solution. A better method is to scan through the result blocks and perform permutations on only the input blocks that contribute to the current result block being processed. This algorithm is shown below.

```
var
    la,lb: array [1..n,1..m,1..nrow,1..ncol] of data;
    lr,lc: array [1..n,1..m,1..nrow,1..ncol] of index;

begin
    for i = 1 to n do
        for j = 1 to m do
            begin {process each result block}
                rb := 1 + lr[i,j] div nrow;
                cb := 1 + lc[i,j] div ncol;
                ro := 1 + lr[i,j] mod nrow;
                co := 1 + lc[i,j] mod ncol;
                for k = 1 to n do
                    for l = 1 to m do
                        begin {consider each input block}
                            maskb := (rb = k) and (cb = l);
                            if any(maskb, 1, 2) then
                                where maskb do
                                    lb[k,l] := perm2 (la [k,l],
                                        ro, co, maskb);
                        end;
            end;
end;
```

Perm2 is the heuristic algorithm presented previously with the modification that the initial mask value is passed as an argument. That is, only elements selected by the mask are permuted. An additional speedup is achieved by this since the heuristic works much better when only a subset of elements are to be permuted.

Table V contains the results of the rotation mapping for the case where a 32 x 32 matrix is considered to consist of 4 x 4 blocks of 8 x 8 elements. The Large Perm results are from using the lrotate approach and the *perm2* results are for the block scanning algorithm.

TABLE V: Comparison of perm2 and large blocked permutation for a rotation centered at coordinates 1 1.

angle of rotation	Perm2		Large perm.	
	rotations	reductions	rotations	reductions
0	13	3	0	0
15	657	703	6768	7184
30	1050	1302	7616	10704
45	1389	1834	8352	13152
60	1702	2329	6896	11248
75	1931	2669	4112	5808
90	2086	2971	1856	1408

TABLE VI: Comparison of perm2 and large blocked permutation for a rotation centered at coordinates 16 16.

angle of rotation	Perm2		Large perm.	
	rotations	comparisons	rotations	comparisons
0	1188	184	3472	1552
15	850	374	7616	7552
30	1011	766	11520	15408
45	1448	1503	16544	25232
60	1858	2174	21328	33552
75	1846	2151	25056	37760
90	1793	2001	22544	33120

MATRIX ROTATION

One application of the permutation function is matrix rotation mapping. Three rotation techniques are considered : nearest neighbor, bilinear interpolation and bicubic interpolation. A rotation is specified by three parameters : the location of the origin of rotation (r_0,c_0) and the rotation angle θ . The starting point of all rotation algorithms is the generation of the mapping matrices r and c from these parameters.

Nearest neighbor

The nearest neighbor algorithm is simply an into mapping in which a result element is assigned the value of the nearest rotated matrix element. In this case the new row and column coordinate matrices, r and c, are defined as follows:

$$r[i,j] = \text{round}((c_0 - j)sin(\theta) + (i - r_0)cos(\theta) + r_0)$$

$$c[i,j] = \text{round}((j - c_0)cos(\theta) + (i - r_0)sin(\theta) + c_0)$$

for all i and j; any values of the result which have near neighbors outside the range of the input matrix are set to zero.

In performing a rotation, some elements of the result matrix are rotated and some elements are selected which are outside of the input matrix. In our algorithm result elements for the latter case are simply set to zero. Therefore we have a permutation in which a subset of the input elements map into a subset of the result elements; the size of these subsets depends upon the angle and origin of the rotation. The rotation is achieved by using the valid elements of r and c with the heuristic permutation algorithm.

Examples of nearest neighbor rotation are shown in Fig. 1. for an 8 x 8 matrix with the origin of rotation located at (1, 1). For a small angle of rotation most of the results have valid mapped values; however, the heuristic algorithm is very effective for this case because there is little movement of the data or *rr* and *cr* are the same for many elements.

When the rotation angle is large there is a lot of data rearrangement but only a few elements are to be moved with the rotation located at (1, 1). For the naive algorithm 49 iterations are needed for all rotations.

Bilinear Interpolation

For the bilinear interpolation algorithm a result element is computed from a weighted sum of the four rotated input matrix elements which surround it. There are two possible approaches to implementing this scheme. First, we can compute four permutations; each permutation acquiring one of the four neighbors for each element. This is called *multiple permutation*.

The second method is to perform one permutation and then seek the local neighborhood of the rotated input matrix for the

11	12	13	14	15	16	17	18
21	22	23	24	25	26	27	28
31	32	33	34	35	36	37	38
41	42	43	44	45	46	47	48
51	52	53	54	55	56	57	58
61	62	63	64	65	66	67	68
71	72	73	74	75	76	77	78
81	82	83	84	85	86	87	88

Input Matrix

11	12	13	0	0	0	0	0
21	22	23	14	15	16	17	18
31	32	33	24	25	26	27	28
42	43	43	34	35	36	37	38
52	53	54	45	46	47	48	0
62	63	64	55	56	57	58	0
72	73	74	65	66	67	68	0
82	83	84	75	76	77	78	0

10 degree rotation

5 iterations

11	0	0	0	0	0	0	0
22	12	0	0	0	0	0	0
22	23	14	0	0	0	0	0
33	24	25	15	0	0	0	0
44	35	25	26	17	0	0	0
55	45	36	27	27	18	0	0
55	56	47	37	28	0	0	0
66	57	57	48	0	0	0	0

45 degree rotation

35 iterations

11	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0

90 degree rotation

14 iterations

Figure 1. Sample Nearest Neighbor Rotations

other near neighbors. The idea being that a local search will require less computation than four complete rotations especially when the angle of rotation is large. The local neighborhood of a single rotated matrix does not contain a complete set of the near-neighbor elements of the input matrix; some are lost due to grid spacing differences. A complete set can be guaranteed, however, if we also include the local neighborhood of a slightly perturbed, rotated input matrix. This scheme is called the *double permutation* method; both rotated matrices can be computed simultaneously with a single execution of a slightly modified heuristic permutation algorithm.

If we map a result element back to the input matrix it will be surrounded by four elements P1 - P4 as shown in Fig. 2. These points are moved to the processing element associated with the result P and the interpolation is then computed in parallel.

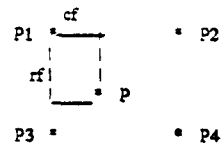

Figure 2. Bilinear interpolation

For the interpolation algorithms, the matrices, rp and cp, contain the actual locations of the rotated elements.

$$rp[i,j] = (c_0 - j)\sin\theta) + (i - r_0)\cos\theta) + r_0$$
$$cp[i,j] = (j - c_0)\cos\theta) + (i - r_0)\sin\theta) + c_0$$

for all i and j.

The coordinates of the near neighbors are as follows:

$$r_{top} = \left| (c_0 - j)\sin\theta) + (i - r_0)\cos\theta) + r_0 \right|$$

$$c_{left} = \left| (j - c_0)\cos\theta) + (i - r_0)\sin\theta) + c_0 \right|$$

$$r_{bottom} = r_{top} + 1$$

$$c_{right} = c_{left} + 1$$

The interpolation fractions are:

$$rf = rp - r_{top}$$

$$cf = cp - c_{left}$$

Once the points P1 - P4 have been obtained the interpolated result is computed as follows:

$$P = (1 - cf)*(1 - rf)*P1 + (1 - rf)*cf*P2 + (1 - cf)*rf*P3 + cf*rf*P4$$

The algorithm for the *multiple permutation* approach is as follows:

```
begin
    r := r_top;
    c := c_left;
    b := coef1 * perm(a, r, c); { value of top left neighbor }
    c := c + 1;
    b := coef2 * perm(a, r, c) + b; { value of top right
    neighbor }
    r := r + 1;
    b := coef3 * perm(a, r, c) + b; { value of bottom right
    neighbor }
    c := c - 1;
    b := coef4 * perm(a, r, c) + b; { value of bottom left
    neighbor }
end;
```

Perm is the heuristic permutation function. The final result of rotating a is stored in the matrix b.

The *double permutation* approach uses a modified permutation function which creates the following matrices:

$$b[i,j] := a[\, r[i,j],\ c[i,j]\,]$$

and

$$d[i,j] := a[\, r[i,j],\ c[i,j] + 1\,]$$

where

a is the original matrix
b is the rotated matrix
d is the shifted rotated matrix
r is the row coordinate matrix

and

c is the column coordinate matrix

To avoid loosing the value of the center of rotation, when the origin is located to the right of the matrix center, the matrix is shifted left and, therefore, in the equation defining matrix d we substitute a negative one for the constant one.

The second step in the *double permutation* algorithm is a local search performed on both rotated and shifted rotated matrices in order to find all the values of the elements needed for the interpolation. The local search has a constant maximum cost for any size matrix. It therefore has an advantage over the *multiple permutations* approach, since every permutation in that approach will become more costly as the matrix size increases.

For the worst case rotation angle ($\theta = 45$), it has been determined that a local search in a 5 x 5 window is sufficient to yield the values of all the elements needed to perform a bilinear interpolation. The local search strategy implemented in our algorithm is a spiral search. The elements are selected by comparing their row and column coordinates to those needed. Once they match, their values can be obtained from the rotated matrix or from the rotated shifted matrix.

Cubic Interpolation

The cubic interpolation version of the rotation algorithm is a simple extension of the bilinear interpolation scheme. The first step, finding the coordinate matrices r and c, is identical to the bilinear interpolation case. After obtaining these matrices, the values of sixteen neighbor points must be acquired. If the *multiple permutations* approach is used, then sixteen separate permutations will be required. However, with the *double permutation* scheme only a small extension of the bilinear algorithm is needed.

Instead of using a 5 x 5 window, which is the case when four points have to be found, a 7 x 7 window is necessary to find sixteen points. However, since the element will have rotated in a specific direction, the search window can be reduced to a 7 x 5 window. Each row of points needed will use a different 7 x 5 window of search.

Once all the values needed are found, the bicubic interpolation itself is done by, first, performing a cubic interpolation for each of the four rows and, then, performing a fifth cubic interpolation on the row points obtained.

As shown in Fig. 3, the reference point for the cubic interpolation computed in step one is P6. The first four cubic interpolations are performed to obtain points pa, pb, pc and pd. The fifth one yields the value of point P.

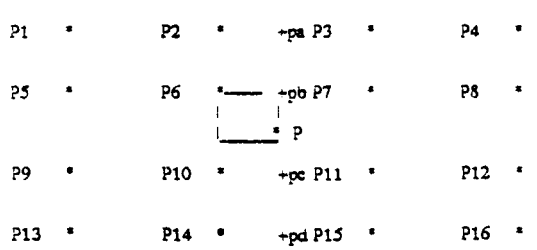

Figure 3. Cubic Interpolation

Test Results

The local search performed in the *double permutation* algorithm has a constant maximum cost for any size matrix. For any matrix the maximum cost is 100 rotations and 25 reductions for the bilinear interpolation method and 552 rotations and 525 comparisons for the cubic interpolation.

The results for rotations using bilinear interpolation for a 32 x 32 matrix are given in Table VII and VIII for different centers of rotation. The results of rotations applying cubic interpolation are given in Tables IX and X.

TABLE VII: Cost of bilinear interpolated rotation centered at coordinates 16 16

angle of rotation	Double perm.		Multiple perm.	
	rotations	reductions	rotations	reductions
0	1188	184	4092	649
15	850	374	2488	1416
30	1011	766	2544	2978
45	1448	1503	3292	5916
60	1858	2174	4084	8602
75	1846	2151	4081	8491
90	1793	2001	4090	7920

TABLE VIII: Cost of bilinear interpolated rotation centered at coordinates 1 1

angle of rotation	Double perm.		Multiple perm.	
	rotations	reductions	rotations	reductions
0	13	3	4	6
15	657	703	1342	2705
30	1050	1302	2035	5093
45	1389	1834	2683	7238
60	1702	2329	3291	9230
75	1931	2669	3471	10627
90	2086	2971	3968	11780

TABLE IX: Cost of cubic interpolated rotation centered at coordinates 16 16

angle of rotation	Double perm.		Multiple perm.	
	rotations	reductions	rotations	reductions
0	1640	684	16368	2608
15	1302	874	10072	5757
30	1463	1266	10394	12021
45	1900	2003	13300	23668
60	2310	2674	16341	34371
75	2298	2651	16315	33928
90	2245	2501	16360	31664

TABLE X: Cost of cubic interpolated rotation centered at coordinates 1 1

angle of rotation	Double perm.		Multiple perm.	
	rotations	reductions	rotations	reductions
0	552	525	272	30
15	1109	1203	7683	11132
30	1502	1802	9723	20589
45	1841	2334	11815	29104
60	2154	2829	13793	37047
75	2383	3169	15316	42586
90	2538	3471	15872	47088

RESULTS USING THE MPP

The results for 32 x 32 matrices reported in this paper were obtained with a Parallel Pascal Translator which translates Parallel Pascal into standard Pascal for program development [4]. Some of these functions have also been run on the MPP; in this case for a 128 x 128 array. In Table XI and Table XII results for near neighbor rotations are given and in tables XIII and XIV results for bilinear interpolation rotations are given.

Table XI: Cost for a near neighbor rotation on a 128 x 128 matrix centered at 1 1.

Angle of rotation	Matrix rotation mapping cost	
	rotations	reductions
0	0	0
15	5114	7506
30	8495	14046
45	11103	20050
60	13504	25121
75	15516	28802
90	16256	32385

In Figs. 4 and 5 comparisons are given between the 32 x 32 and 128 x 128 results. The cost shown is the ratio of the number of shift operations required by the heuristic algorithm over the number of shifts required by the simple algorithm for a single permutation (i.e., n^2). These figures show that there is a very good correspondence between the the results for the different size matrices. That is, for the rotation algorithm the improvement achieved with the heuristic algorithm is a constant which which is independent of matrix size.

Table XII: Cost for a near neighbor rotation on a 128 x 128 matrix centered at 64 64.

Angle of rotation	Matrix rotation mapping cost	
	rotations	reductions
0	0	0
15	9183	3594
30	10446	7800
45	13589	15777
60	16378	23359
75	16375	24331
90	16319	16384

Table XIII: Cost for a bilinear interpolation rotation on a 128 x 128 matrix centered at 1 1.

Angle of rotation	Matrix rotation mapping cost	
	rotations	reductions
0	101	25
15	9009	7555
30	15689	14107
45	21246	20034
60	26228	21192
75	30023	28893
90	32614	32410

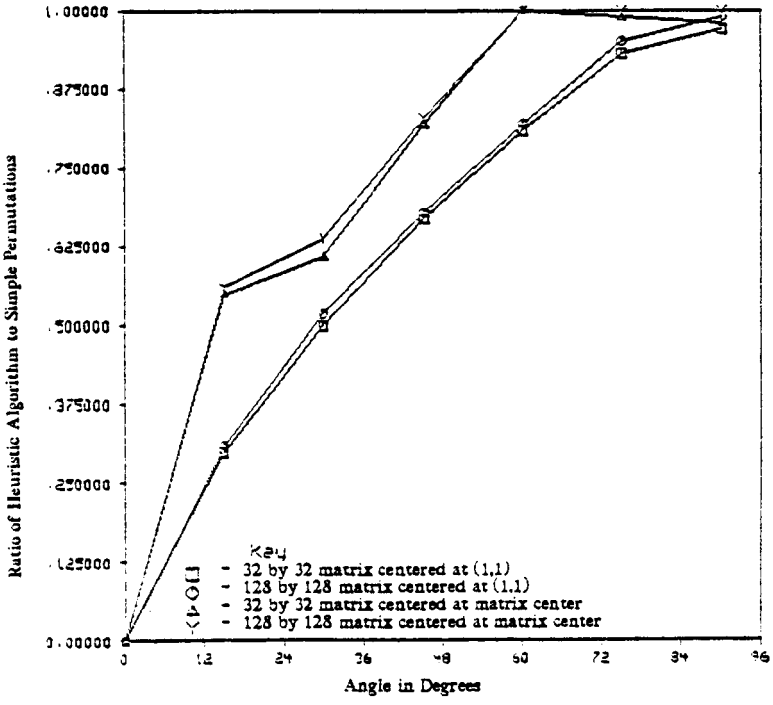

Figure 4. Ratio of the number of rotations to n^2 rotations for near neighbor rotation with the center of rotation at (1,1) and at the matrix center.

Table XIV: Cost for a bilinear interpolation rotation on a 128 x 128 matrix centered at 64 64.

Angle of rotation	Matrix rotation mapping cost	
	rotations	reductions
0	101	25
15	11510	3628
30	14648	7826
45	21556	15800
60	28282	23378
75	28763	24382
90	24676	16409

CONCLUSION

An effective heuristic algorithm for arbitrary permutations and data mappings for mesh connected SIMD processors has been presented. This algorithm is particularly suited to the following conditions:

1. When only a few elements are to be moved.

2. When many elements share a similar motion, e.g. small angle matrix rotation and warping.

3. When large arrays are to be processed.

It is less suitable when the permutation or mapping is dense and does not have the locality property. The effectiveness of this algorithm over a naive algorithm depends upon the system implementation parameters, and the size of the data to be manipulated.

An effective technique for matrix rotation interpolation has been presented which involves a local search scheme. Excellent results have been obtained especially for bicubic interpolation.

REFERENCES

1. K. E. Batcher, "Design of a Massively Parallel Processor," *IEEE Transactions on Computers* C-29(9) pp. 836-840 (September 1981).

2. A. P. Reeves, "Parallel Pascal: An extended Pascal for Parallel computers," *Journal of Parallel and Distributed Computing* 1 pp. 64-80 (1984).

3. A. P. Reeves, "On Efficient Global Information Extraction Methods For Parallel Processors," *Computer Graphics and Image Processing* 14 pp. 159-169 (1980).

4. A. P. Reeves, "Parallel Pascal Development System," *Cornell University Technical Report*, (January, 1985).

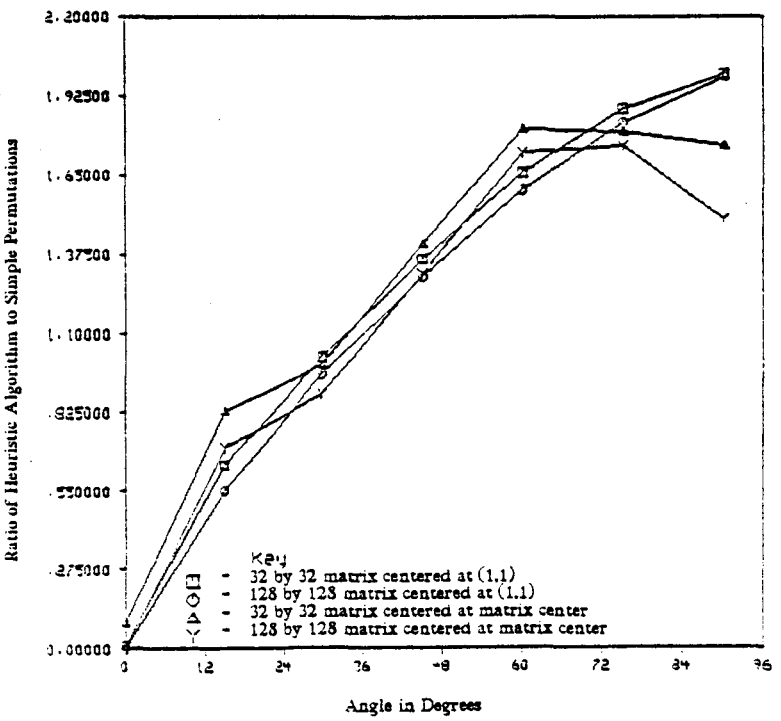

Figure 5. Ratio of the number of rotations to n^2 rotations for bilinear interpolation rotation with the center of rotation at (1,1) and at the matrix center.